07-03-00

A

jc832 U.S. PTO 09/609494 06/30/00

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent Application of: Manuvir Das et al.
Title: METHODS FOR ENHANCING FLOW ANALYSIS
Attorney Docket No.: 777.383US1

PATENT APPLICATION TRANSMITTAL

BOX PATENT APPLICATION
Commissioner for Patents
Washington, D.C. 20231

We are transmitting herewith the following attached items and information (as indicated with an "X"):

X Return postcard.
X Utility Patent Application under 37 CFR § 1.53(b) comprising:
  X Specification (40 pgs, including claims numbered 1 through 56 and a 1 page Abstract).
  X Formal Drawing(s) (7 sheets).
  X Signed Combined Declaration and Power of Attorney (4 pgs).
  X Check in the amount of $2,508.00 to pay the filing fee.
X Assignment of the invention to Microsoft Corporation (3 pgs) and Recordation Form Cover Sheet.
X Check in the amount of $40.00 to pay the Assignment recording fee.
X Information Disclosure Statement (1 pgs), Form 1449 (1 pgs), and copies of cited references (1).

The filing fee has been calculated below as follows:

	No. Filed	No. Extra	Rate	Fee
TOTAL CLAIMS	56 - 20 =	36	x 18 =	$648.00
INDEPENDENT CLAIMS	18 - 3 =	15	x 78 =	$1,170.00
[ ] MULTIPLE DEPENDENT CLAIMS PRESENTED				$0.00
BASIC FEE				$690.00
TOTAL				$2,508.00

Please charge any additional required fees or credit overpayment to Deposit Account No. 19-0743.

SCHWEGMAN, LUNDBERG, WOESSNER & KLUTH, P.A.
P.O. Box 2938, Minneapolis, MN 55402 (612-373-6900)

Customer Number 21186

By:
Atty: ~~D. C.~~ Peter Chu
Reg. No. 41,676

"Express Mail" mailing label number: EL618476445US
Date of Deposit: June 30, 2000
I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to the Commissioner for Patents, Box Patent Application, Washington, D.C. 20231.

By: Shawn L. Hise
Signature:

UNITED STATES PATENT APPLICATION

METHODS FOR ENHANCING FLOW ANALYSIS

INVENTORS

Manuel Fähndrich

Jakob Rehof

Manuvir Das

Schwegman, Lundberg, Woessner, & Kluth, P.A.

1600 TCF Tower

121 South Eighth Street

Minneapolis, Minnesota 55402

ATTORNEY DOCKET 777.383US1

MICROSOFT 150406.1

DRAFTING ATTORNEY

Peter Chu

METHODS FOR ENHANCING FLOW ANALYSIS

Technical Field

The technical field relates generally to program analysis. More particularly, it pertains to flow analysis of programs that include function pointers.

Copyright Notice - Permission

Background

A program is a list of statements. This list of statements may be translated, through processes that include compilation, to produce an executable file that can cause a computer to perform a desired action. One type of statement is an assignment statement. An illustrative example of an assignment statement is x=y(). This statement may be translated to mean that the result of the invocation of the function variable y is assigned to the variable x.

One type of variable is a function pointer. Pointers are often used in programs because they offer flexibility in coding. A function is a group of statements identified by a name that can be invoked within a program by referencing that name. A function pointer contains a location of a function. Thus, a function pointer points to a function. Through a function pointer, a function may be invoked. In the idiom of software analysis, programs that have function

pointers are called higher-order programs, and programs that lack function pointers are called first-order programs.

It is beneficial to analyze programs in order to obtain information that may be used to improve them. One type of analysis focuses on the flow of values in a program. Values arise from various computations in a program to obtain a desired result. Because of the presence of functions within a program, values flow in and out of those functions. The type of analysis that analyzes the flow of values in a program to provide an understanding of the behavior of the program is known as flow analysis. Flow analysis is complicated by the presence of function pointers.

Current flow analyses inefficiently deal with the presence of function pointers. One particular technique is described by the following references: See Ramkrisha Chatterjee et al., Relevant Context Inference, Conference Record of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (January 1999); Donglin Liang and Mary Jean Harrold, Efficient Points-to Analysis for Whole-Program Analysis, Proceedings of the 7th European Software Engineering Conference And the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (September 1999). Another similar technique is described by the following references: See John Whaley and Martin Rinard, Compositional Pointer And Escape Analysis for Java Programs, OOPSLA, pp. 187-206 (1999); Jong-Deok Choi et al., Escape Analysis for Java, OOPSLA, pp. 1-19 (1999). A different technique is described by the following references: See Robert P. Wilson and Monica S. Lam, Efficient Context-Sensitive Pointer Analysis for C Programs, Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation (June 1995). Current flow analyses described by the above techniques are either too costly in terms of time or too imprecise in terms of information. Tools that rely on such pointer analyses such as optimizer and debugging tools have been constrained by having to make inferior assumptions about behaviors of programs.

As the size of programs has increased with each generation of technology, such inferior assumptions may slow the improvement of programs and lead to the eventual lack of acceptance of such programs in the marketplace.

Thus, what is needed are methods and structures to enhance flow analysis of programs.

Summary

Methods and structures to support enhanced flow analysis are discussed. An illustrative aspect includes a method for enhancing flow analysis. The method includes inferring types for a program, forming a type graph having polarities and indices from the types, and forming a flow graph from the type graph to inhibit imprecise paths so as to enhance context-sensitivity of flow analysis. Another illustrative aspect includes another method for enhancing flow analysis. The method includes forming a type graph that includes polarities and indicies, and forming a flow graph that includes a set of flow paths. The set of flow paths excludes imprecise paths so as to enhance context-sensitivity of flow analysis. Another illustrative aspect includes another method for enhancing flow analysis. The method includes abstracting program expressions into types, and forming a type graph from the types. The type graph includes polarities and indices so as to enhance flow analysis. Another illustrative aspect includes another method for enhancing flow analysis. The method includes annotating each expression in a program by a label, associating the label of an expression with a type of the expression, and tracing at least one path on a type graph having polarities to determine if a value arising at one label in the program flows to another label in the program. Yet another illustrative aspect includes another method for enhancing flow analysis. The method includes forming a type instantiation graph that includes polarities and indices, and computing points-to information for at least one program point by answering reachability queries on the type instantiation

graph. The polarities of the type graph as discussed hereinbefore exist on a plurality of instantiation edges of the type graph, in one aspect.

Another illustrative aspect includes a method for inferring types to enhance flow analysis. The method includes generating constraints from a program and solving the constraints to infer at least one type. The act of solving includes propagating polarities so as to enhance flow analysis.

Another illustrative aspect includes a data structure to enhance flow analysis. The data structure includes a data member type to represent a type of a program expression, and a data member flow having a data member polarity and a data member index to represent a flow path between two types.

Brief Description of the Drawings

Figure 1 is a block diagram of a system according to one aspect of the present invention.

Figure 2 is a fragment of a program according to one aspect of the present invention.

Figures 3A, 3B, and 3C illustrate a sequence of processing according to one aspect of the present invention.

Figures 4A, 4B, and 4C illustrate a sequence of processing according to one aspect of the present invention.

Figure 5 is a process diagram of a method according to one aspect of the present invention.

Figure 6 is a process diagram of a method according to one aspect of the present invention.

Figure 7 is a structure diagram of a data structure according to one aspect of the present invention.

Figure 8 is a process diagram of a method according to one aspect of the present invention.

Figure 9 is a block diagram of a system according to one aspect of the present invention.

Detailed Description

In the following detailed description of exemplary embodiments of the invention, reference is made to the accompanying drawings which form a part hereof, and in which is shown, by way of illustration, specific exemplary embodiments in which the invention may be practiced. In the drawings, like numerals describe substantially similar components throughout the several views. These embodiments are described in sufficient detail to enable those skilled in the art to practice the invention. Other embodiments may be utilized and structural, logical, electrical, and other changes may be made without departing from the spirit or scope of the present invention. The following detailed description is, therefore, not to be taken in a limiting sense, and the scope of the present invention is defined only by the appended claims.

Figure 1 is a block diagram of a system according to one aspect of the present invention. Figure 1 provides a brief, general description of a suitable computing environment in which the invention may be implemented. The invention will hereinafter be described in the general context of computer-executable program modules containing instructions executed by a personal computer (PC). Program modules include routines, programs, objects, components, data structures, etc., that perform particular tasks or implement particular abstract data types. Those skilled in the art will appreciate that the invention may be practiced with other computer-system configurations, including hand-held devices, multiprocessor systems, microprocessor-based programmable consumer electronics, network PCs, minicomputers, mainframe computers, and the like, which have multimedia capabilities. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices linked through a communications network. In a distributed

computing environment, program modules may be located in both local and remote memory storage devices.

Figure 1 shows a general-purpose computing device in the form of a conventional personal computer 120, which includes processing unit 121, system memory 122, and system bus 123 that couples the system memory and other system components to processing unit 121. System bus 123 may be any of several types, including a memory bus or memory controller, a peripheral bus, and a local bus, and may use any of a variety of bus structures. System memory 122 includes read-only memory (ROM) 124 and random-access memory (RAM) 125. A basic input/output system (BIOS) 126, stored in ROM 124, contains the basic routines that transfer information between components of personal computer 120. BIOS 126 also contains start-up routines for the system. Personal computer 120 further includes hard disk drive 127 for reading from and writing to a hard disk (not shown), magnetic disk drive 128 for reading from and writing to a removable magnetic disk 129, and optical disk drive 130 for reading from and writing to a removable optical disk 131 such as a CD-ROM or other optical medium. Hard disk drive 127, magnetic disk drive 128, and optical disk drive 130 are connected to system bus 123 by a hard-disk drive interface 132, a magnetic-disk drive interface 133, and an optical-drive interface 134, respectively. The drives and their associated computer-readable media provide nonvolatile storage of computer-readable instructions, data structures, program modules, and other data for personal computer 120. Although the exemplary environment described herein employs a hard disk, a removable magnetic disk 129 and a removable optical disk 131, those skilled in the art will appreciate that other types of computer-readable media which can store data accessible by a computer may also be used in the exemplary operating environment. Such media may include magnetic cassettes, flash-memory cards, digital versatile disks, Bernoulli cartridges, RAMs, ROMs, and the like.

Program modules may be stored on the hard disk, magnetic disk 129, optical disk 131, ROM 124, and RAM 125. Program modules may include operating system 135, one or more application programs 136, other program modules 137, and program data 138. A user may enter commands and information into personal computer 120 through input devices such as a keyboard 140 and a pointing device 142. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 121 through a serial-port interface 146 coupled to system bus 123; but they may be connected through other interfaces not shown in Figure 1, such as a parallel port, a game port, or a universal serial bus (USB). A monitor 147 or other display device also connects to system bus 123 via an interface such as a video adapter 148. In addition to the monitor, personal computers typically include other peripheral output devices such as a sound adapter 156, speakers 157, and additional devices such as printers.

Personal computer 120 may operate in a networked environment using logical connections to one or more remote computers such as remote computer 149. Remote computer 149 may be another personal computer, a server, a router, a network PC, a peer device, or other common network node. It typically includes many or all of the components described above in connection with personal computer 120; however, only a storage device 150 is illustrated in Figure 1. The logical connections depicted in Figure 1 include local-area network (LAN) 151 and a wide-area network (WAN) 152. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet.

When placed in a LAN networking environment, PC 120 connects to local network 151 through a network interface or adapter 153. When used in a WAN networking environment such as the Internet, PC 120 typically includes modem 154 or other means for establishing communications over network 152. Modem

154 may be internal or external to PC 120, and connects to system bus 123 via serial-port interface 146. In a networked environment, program modules, such as those comprising Microsoft® Word which are depicted as residing within PC 120 or portions thereof may be stored in remote storage device 150. Of course, the network connections shown are illustrative, and other means of establishing a communications link between the computers may be substituted.

Software may be designed using many different methods, including object-oriented programming methods. C++ is one example of common object-oriented computer programming languages that provide the functionality associated with object-oriented programming. Object-oriented programming methods provide a means to encapsulate data members (variables) and member functions (methods) that operate on that data into a single entity called a class. Object-oriented programming methods also provide a means to create new classes based on existing classes.

An object is an instance of a class. The data members of an object are attributes that are stored inside the computer memory, and the methods are executable computer code that acts upon this data, along with potentially providing other services. The notion of an object is exploited in the present invention in that certain aspects of the invention are implemented as objects in one embodiment.

An interface is a group of related functions that are organized into a named unit. Each interface may be uniquely identified by some identifier. Interfaces have no instantiation, that is, an interface is a definition only without the executable code needed to implement the methods which are specified by the interface. An object may support an interface by providing executable code for the methods specified by the interface. The executable code supplied by the object must comply with the definitions specified by the interface. The object may also provide additional methods. Those skilled in the art will recognize that interfaces are not limited to use in or by an object-oriented programming environment.

The embodiments of the present invention focus on enhancing flow analysis. The embodiments of the present invention present a framework that analyzes a program that may include function pointers. In the various embodiments, function pointers are treated as if they were any other program expressions. The embodiments of the present invention need not enumerate all the potential functions that may be referenced by a function pointer. The need to enumerate all possibilities may cripple a flow analysis of a large program.

The embodiments of the present invention also enhance the context-sensitivity of flow analysis. Context-sensitivity means the inclusion of keeping a function invocation distinct from another invocation of the same function. This ability to distinguish occurrences of function invocation allows the embodiments of the present invention to have a desired analytical precision within a desired duration of analysis. The following discusses the embodiments of the present invention in more detail.

Figure 9 is a block diagram of a system according to one aspect of the present invention. The system 900 includes a source file 902. The source file 902 includes a program that will be analyzed by the system 900. The source file 902 is input into a flow-analysis engine 904. The flow-analysis engine 904 is context-sensitive. In one embodiment, the flow-analysis engine 904 is a piece of software executing on a suitable computing platform. The flow-analysis engine 904 produces flow information 906 that excludes imprecise paths. Imprecise paths would render a flow analysis context-insensitive.

Figure 2 is a fragment of a program according to one aspect of the present invention. A fragment 200 of a program includes an assignment statement 202. The assignment statement 202 includes a pointer variable p being assigned an address of a variable x. The fragment 200 includes an assignment statement 204. The assignment statement 204 includes a pointer variable q being assigned an address of a variable y.

The fragment 200 includes an assignment statement 206. The assignment statement 206 includes the pointer variable p being assigned the return value of a function id(). The function id() takes the pointer p as an input argument. The fragment 200 includes an assignment statement 208. The assignment statement 208 includes the pointer variable q being assigned the return value of a function id(). The function id() takes the pointer q as an input argument.

The fragment 200 includes a function id() 210. The function id() 210 is identified by the name "id." The function id 210 includes an input parameter. The input parameter is a pointer to an integer. The input parameter is identified by the name "c." The function id() 210 includes an opened bracket 212 and a closed bracket 216. The opened bracket 112 and the closed bracket 216 define a scope for the function id() 210. Thus, statements that exist inside the scope of the function id() 210 are considered to be the group of statements that may be executed upon an invocation of the function id() 210. The function id() 210 includes a return statement 214. The return statement 214 returns the value of the integer pointer variable c.

In a normal sequence of execution of the fragment 200, the assignment statement 206 invokes the function id() 210, which is represented by flow path 218. The value of the pointer variable p traverses the flow path 218 from the assignment statement 206 and enters the function id() 210 through the input parameter c. Next, the output of the function id() 210, which is the value of the integer pointer variable c, traverses the flow path 222 to the assignment 206. The value of the integer pointer variable c is then assigned to the pointer variable p. Similarly, the assignment statement 208 invokes the function id() 210, which is represented by flow path 220. The value of the pointer variable q traverses the flow path 220 from the assignment statement 208 and enters the function id() 210 through the input parameter c. Next, the value of the integer pointer variable c traverses the flow path 224 to the assignment 208. The value of the integer

pointer variable c is then assigned to the pointer variable q.

Without at least one of the embodiments of the present invention, a possibility exists that a static analysis of a value of the pointer p traversing the flow path 218 by an invocation of the function id() 210 may return via the flow path 224, or similarly, a value of the pointer q traversing the flow path 220 may return via the flow path 222. Such undesirable flows of values render a flow analysis to be context-insensitive. The undesired paths in which the values flow are defined as imprecise paths. Because the fragment 200 benefits from at least one of the embodiments of the present invention, the discussed imprecise paths are eliminated. Thus, the embodiments of the present invention enhance the context-sensitivity of flow analysis.

Figures 3A, 3B, and 3C illustrate a sequence of processing according to one aspect of the present invention. Through these Figures, various embodiments of the present invention will show that function pointers are treated as if they were any other program expressions that are analyzed in a program analysis. Figure 3A illustrates a fragment 300 of a program according to one aspect of the present invention. The fragment 300 includes a declaration statement 302. The declaration statement 302 defines a new data type through the "typedef" mechanism. The new data type defined by the declaration statement 302 is identified as "FIP," which is a pointer to a function that takes an integer pointer and returns a void.

The fragment 300 includes a function definition 304. The function definition 304 defines a function that is identified as "f," which is a function that takes an integer pointer and returns a void. The body of the function definition 304 is filled with an ellipsis to indicate that the statements therein are not needed to discuss of the embodiments of the present invention.

The fragment 300 includes a comment 306 delimited by the symbols "/*" and "*/". The comment 306 includes a type expression. Hereinafter, the type

expressions in the comments are those after the colons. The symbols before the colons are the identifiers for program expressions for which the type expressions are inferred.

The comment 306 includes inferred types for the signature of the function f as defined in the function definition 304. These inferred types arise from the type-based flow analysis of the embodiments of the present invention. The terms "type" and "type expression" can be used interchangeably in the embodiments of the present invention. Type-based flow analysis assigns types and locations to program expressions. It should be understood that the embodiments of the present invention could be applied to any type languages, including, but not limited to C, C++, or Java.

For the purpose of the fragment 300, three types may be defined to analyze the fragment 300. In one embodiment, these three types may be mathematically described as follows:

$$(1)\ \tau ::= \alpha \mid (\tau_1, \ldots, \tau_n) \to^{\ell} \tau \mid ptr^{\ell}(\tau)$$

τ represents a type. α represents a type variable or an unknown type. $(\tau_1, \ldots, \tau_n) \to^{\ell} \tau$ represents a type for a function signature. The $(\tau_1, \ldots, \tau_n)$ aspect of the type for the function signature represents the input parameters of a function. The $\to^{\ell}$ aspect of the type for the function signature represents a type that maps the $(\tau_1, \ldots, \tau_n)$ aspect to a result type. The τ aspect of the type for the function signature represents the result type. The $ptr^{\ell}(\tau)$ represents a type for a pointer pointing to an expression of type τ. ℓ is a flow variable. Flow variables are used to uniquely name program expressions of interest, such as pointers, functions, and locations. For example, in the type $(\tau_1, \ldots, \tau_n) \to^{\ell} \tau$, ℓ is the location of a particular function, and in the type $ptr^{\ell}(\tau)$, ℓ is a location named ℓ, such that $ptr^{\ell}(\tau)$ is a pointer to the location names ℓ. Furthermore, $[\tau]^{\ell}$ represents a memory location named ℓ that holds values of type τ. Suppose that the programming language of interest is C. Then, $[\tau]^{\ell}$ is associated with L-values and τ is associated

with R-values.

Returning to Figure 3A, the signature of the function f at function definition 304 gives rise to the inferred types shown in the comment 306.

The fragment 300 includes a function definition 308 that defines the function "g." The function g takes void as an input parameter and returns a result type of type FIP as defined in the declaration statement 302. The comment 310 includes an inferred type for the function signature of the function g. The fragment 300 includes a return statement 312. The return statement 312 returns the address of the function f as defined in statement 304. Therefore, the function g returns a function pointer that contains an address of the function f.

The fragment 300 includes a comment 314. The comment 314 includes an instantiation of the type expression for the function f. An instantiation of a type expression is another type. The generic type for such an instantiation is as shown in the comment 306. This generic type is the inferred type for the function signature for the function f. The type that is instantiated from the generic type is known as an instance type. This instance type includes an index i as shown in the comment 306 so as to track a particular occurrence of an invocation of the function f. The numerical references of the flow labels ℓ of the instance type as shown in the comment 314 are different with respect to numerical references of the flow labels ℓ of the generic type as shown in the comment 306. This is due to the process of instantiation so as to differentiate various instances of the generic type.

The fragment 300 includes a function definition 316 for a function identified as "h." The function h includes a declaration statement 318. The declaration statement 318 declares the variable "c" as an integer type. The fragment 300 includes a comment 320 that includes an inferred type for the variable c.

The function h includes a statement 322. The statement 322 is a

combination of a declaration statement, a function invocation, and an assignment statement. The statement 322 as a declaration statement declares that the variable fp is of a type FIP. Therefore, fp is a function pointer that points to a function that takes an integer pointer as a parameter and returns type void. The statement 322 also invokes the function g. The statement 322 then assigns the result of the invocation of the function g and assigns the result to the variable fp. As discussed above, the function g returns the location of the function f. Therefore, after execution of the statement 322, the variable fp is a function variable that contains the location of the function f.

The fragment 300 includes a comment 324. The comment 324 includes an instantiation of the generic type for the function signature for the function g as shown in the comment 310. The comment 324 includes an index j to denote an occurrence of an invocation of the function g at the statement 322.

The fragment 300 includes a function invocation statement 326. The function invocation statement 326 invokes the function that is pointed to by the function pointer fp. That function is the function f.

The fragment 300 includes a comment 328. The comment 328 includes the result type of the instance type for the function signature for the function g as shown in the comment 324.

What has been shown in Figure 3A are acts that transform program expressions into types, for illustrative purposes only. These types are shown in the comments 306, 310, 314, 320, 324, and 326. At least one type of these types may be instantiated from another type to denote an occurrence of a use of a program expression as represented by the instantiated type. Such an instantiation economizes the flow analysis while including information regarding the context-sensitivity. Also what has been shown in Figure 3A is that function pointers, such as fp in the fragment 300, are transformed into types as if they were any other program expressions.

Figure 3B shows a graph following the next sequence of processing. A type instantiation graph 330 (hereinafter, the type graph 330) is formed from a portion of the types inferred from the fragment 300. In one embodiment, the type graph 330 represents a complete trace of all instantiations. The type graph 330 includes a number of nodes to represent inferred types. A node 332 represents the inferred type for the function f as shown in the statement 304. A node 334 represents an inferred type for the variable p as shown in statement 304. A node 336 represents a variable that is pointed to by the variable p. The lines 338 and 340 couple the nodes 332, 334, and 336 to show that the types represented by these nodes are related to each other. Nodes 332, 334, and 336 are generated from inferred types from the function signature of the function f as shown in statement 304. The inferred types are shown in the comment 306.

Nodes 348, 350, and 352 are generated from inferred types that are generated from statement 312. The inferred types are shown in the comment 314. The lines 354 and 356 couple the nodes 348, 350, and 352 to show that the types represented by these nodes are related to each other. In one embodiment, each of the nodes 348, 350, and 352 can be viewed as types instantiated from nodes 332, 334, and 336, respectively.

Such instantiations are represented by the instantiation paths 342, 344, and 346. An instantiation path emanates from a generic type and terminates at an instance type. The instantiation path includes an arrowhead to show an instantiation direction. The instantiation path includes an instantiation constraint that is symbolized by $\leq^{index}_{polarity}$. This instantiation constraint is an inequality to represent that a type is an instance of another type. The inequality includes an index and a polarity. The index represents an occurrence of the instantiation, or more specifically, an occurrence of the use of a program expression. The polarity can be a 0, which is a positive polarity, or a 1, which is a negative polarity, or a T, which is a bidirectional polarity.

Nodes 364, 366, and 368 are generated from inferred types that are generated from statement 322. The inferred types are shown in the comments 324 and 328. The lines 370 and 372 couple the nodes 364, 366, and 368 to show that the types represented by these nodes are related to each other. In one embodiment, each of the nodes 364, 366, and 368 can be viewed as types instantiated from nodes 348, 350, and 352, respectively.

Figure 3C shows a graph following the next sequence of processing. A flow graph 374 is formed from the type graph 330. In one embodiment, the flow graph 374 retains the nodes of the type graph 330. The flow graph 374 includes flow paths 342_0 and 358_0. The flow paths 342_0 and 358_0 emanate from the generic types and terminate at the instance types. Thus, the flow paths 342_0 and 358_0 have the same flow direction as the instantiation direction of the instantiation paths that relate the nodes 332, 348, and 364 together. Therefore, in one embodiment, it can be considered that the flow path between two nodes inherits the polarity from the instantiation path that relates the same two nodes. If the polarity is positive (or 0), the flow direction is the same as the instantiation direction. The flow graph 374 includes the flow paths 344_1 and 360_1. The polarity of the flow paths 344_1 and 360_1 are negative (or 1), and thus, the flow direction of these flow paths are opposite the instantiation direction of the corresponding instantiation paths 344 and 360. The flow graph 374 includes flow paths 346_0, 346_1, 362_0, and 362_1. These flow paths arise from the bidirectional polarity of the instantiation paths 346 and 362.

Figures 4A, 4B, 4C illustrate a sequence of processing according to one aspect of the present invention. Figure 4A illustrates a fragment 400 of a program according to one aspect of the present invention. Through these Figures, various embodiments of the present invention will show that imprecise paths are identified and eliminated or inhibited so as to enhance context-sensitivity of flow analysis. The fragment 400 is used for illustrative purposes only. The fragment 400

includes a function definition statement 402. The function definition statement 402 defines a function id. The function id is a function that takes a pointer to an integer and returns a pointer to an integer. The fragment 400 includes a comment 404. The comment 404 shows the inferred types for the function signature of the function id.

The fragment 400 includes a function definition statement 406 to define a function foo. The definition of the function foo includes a declaration statement 408. The declaration statement 408 declares a variable b as an integer. The fragment 400 includes a comment 410. The comment 410 shows the inferred type for the declaration of the variable b. The definition of the function foo includes a function invocation statement 412. The function invocation statement 412 invokes the function id with the address of b as the input argument. The fragment 400 includes a comment 414 that shows the inferred types for the invocation of the function id at statement 412.

The fragment 400 includes a function definition statement 416 to define a function bar. The definition of the function bar includes a declaration statement 418. The declaration statement 418 declares a variable c as an integer. The fragment 400 includes a comment 420. The comment 420 shows the inferred type for the declaration of the variable c. The definition of the function bar includes a function invocation statement 422. The function invocation statement 422 invokes the function id with the address of c as the input argument. The fragment 400 includes a comment 424 that shows the inferred types for the invocation of the function id at statement 422.

Figure 4B shows a graph following the next sequence of processing. A type instantiation graph 466 (hereinafter, the type graph 466) is formed from a portion of the types inferred from the fragment 400. The nodes 426, 430, and 434 are formed from the inferred types as indicated in the comment 404. The nodes are also related through the lines 428_a, 428_b, and 432. These lines 428_a, 428_b, and

432 represent the relationship between the types as inferred from the function signature of the function id.

The nodes 436, 440, and 444 are formed from the inferred types as indicated in the comment 414. The nodes are also related through the lines 438_a, 438_b, and 442. These lines 438_a, 438_b, and 442 represents the relationship between the types as inferred from the function signature of the function id. In one embodiment, the nodes 436, 440, and 444 represent instance types of the generic types as represented by nodes 426, 430, and 434. The instantiation paths 455, 457, and 459 represent the instantiations of these instance types.

The nodes 446, 450, and 454 are formed from the inferred types as indicated in the comment 424. The nodes are also related through the lines 448_a, 448_b, and 452. These lines 448_a, 448_b, and 452 represent the relationship between the types as inferred from the function signature of the function id. In one embodiment, the nodes 446, 450, and 454 represent instance types of the generic types as represented by nodes 426, 430, and 434. The instantiation paths 461, 463, and 465 represent the instantiations of these instance types.

Figure 4C shows a graph following the next sequence of processing. A flow graph 468 is formed from the type graph 466. The flow graph 468 includes flow paths 455_0 and 461_0. The flow graph 468 includes flow paths 459_0, 459_1, 465_0, and 465_1. The flow paths 459_0, 459_1, 465_0, and 465_1 are formed from the bidirectional polarity of the instantiation paths 459 and 465. The flow graph 468 includes flow paths 457_0, 457_1, 463_0, and 463_1. The flow paths 457_0, 457_1, 463_0, and 463_1 are formed from the bidirectional polarity of the instantiation paths 457 and 463.

The flow graph 468 includes at least one imprecise path. For illustrative purposes only, suppose a question is posed to a flow analysis regarding what could be returned at the invocation of the function id at statement 422. Without the use of at least one embodiment of the present invention, the answer to that question

would include a pointer to location ℓ_3 and a pointer to location ℓ_5 via the path 457_1 and 463_0. Such a path is an imprecise path because it traces a flow from the invocation of the function id at statement 412, through the definition of the function id at statement 402, and returning to the site of another invocation of the function id at the statement 422. Such an imprecise path corrupts the flow analysis and renders such an analysis context-insensitive.

In one embodiment, at least one imprecise path is inhibited. In another embodiment, at least one imprecise path is eliminated. In another embodiment, the flow analysis considers a set of paths that excludes at least one imprecise path so as to enhance context-sensitivity of the flow analysis. In one embodiment, the imprecise path includes an edge that has a negative polarity (1 polarity) preceding an edge that has a positive polarity (0 polarity). In one embodiment, the imprecise path includes a path that includes a flow of a parameter from an invocation of a function and a flow of a result of a function back to another invocation of the function. In one embodiment, the flow analysis considers a set of paths where each path begins with any number of edges of positive polarity and is followed by any number of edges of negative polarity. In one embodiment, a path, as used hereinbefore and hereinafter, means the inclusion of a sequence of edges. In another embodiment, the path means the inclusion of at least one edge.

Figure 5 is a process diagram of a method according to one aspect of the present invention. A process 500 is a method for enhancing flow analysis. The process 500 includes an act 502 for inferring types from a program, an act 504 for forming a type graph from the types, and an act 506 for forming a flow graph from the type graph to inhibit imprecise paths so as to enhance context-sensitivity of flow analysis.

The act 502 for inferring types includes an act for generating constraints from the program and solving the constraints. The constraints include a set of equalities and inequalities. In one embodiment, the set is a finite set. In another

embodiment, the set is adapted to be a set of simultaneous equations. An equality from the set of equalities defines that a type is equal to another type such that the type and the another type are adapted to be unified. The process of unification is discussed by Bjarne Steensgaard, Points-to Analysis In Almost Linear Time, Conference Record of the Twenty-Third ACM Symposium on Principles of Programming Languages, p. 32-41 (January 1996). Such process of unification does not limit the embodiments of the present invention, and as such, will not be presented here in full. An inequality from the set of inequalities defines that a type is an instance of another type. The inequality includes an instantiation constraint. The instantiation constrain includes an index and a polarity.

In one embodiment, the method for enhancing flow analysis includes an act for forming a type graph that includes polarities and indices and an act for forming a flow graph that includes a set of flow paths. The set of flow paths excludes imprecise paths so as to enhance context-sensitivity of flow analysis. The act of forming the set of flow paths includes forming at least one flow path that inherits a polarity from the polarities of the type graph. The imprecise path includes two flow edges; the polarity of one of the two flow edges is negative and the polarity of the other of the two flow edges is positive. The one of the two flow edges preceding the other of the two flow edges defines the imprecise path. In one embodiment, each polarity indicates at least one of a value flowing into a type and a value flowing out of a type.

In another embodiment, the method for enhancing flow analysis includes an act for abstracting program expressions into types and forming a type graph from the types. The act of abstracting allows the flow analysis to track the flow at the level of types instead of at the level of values. The type graph includes polarities and indices so as to enhance flow analysis. The act of forming includes forming a plurality of nodes. Each node represents a type. The act of forming includes forming at least one instantiation path between two nodes to represent an

instantiation of a generic type to an instance type. The instantiation path includes an index and a polarity. The act of abstracting includes unifying recursive types. The act of unifying is adapted to be executed using a cyclic unification technique.

In another embodiment, the method for enhancing flow analysis includes an act for annotating each expression in a program by a label. The method includes an act for associating the label of an expression with a type of the expression. The method also includes an act for tracing at least one path on a type graph having polarities to determine if a value arising at one label in the program flows to another label in the program.

In another embodiment, the method for enhancing flow analysis includes an act for forming a type instantiation graph that includes polarities and indices and an act for computing points-to information for at least one program point by answering reachability queries on the type instantiation graph. The act of forming includes forming with a polymorphic inference technique based on instantiation constraints. In one embodiment, polymorphism means the inclusion of context-sensitivity with respect to types. In another embodiment, context-sensitivity means that a program expression, such as a function, can be instantiated to different types in different contexts if the program expression is of a particular polymorphic type. The act of answering includes answering the reachability queries within a desired duration. The desired duration is linearly proportional to a size of the type of the instantiation graph.

In another embodiment, the method for enhancing flow analysis includes an act for forming a type instantiation graph that includes polarities and indices. The method includes an act for forming a flow graph to form a set of flow paths. The method further includes an act for forming a subset from the set of flow paths to enhance the context-sensitivity of flow analysis. The set of flow paths includes at least one imprecise path. The subset excludes the at least one imprecise path.

Figure 6 is a process diagram of a method according to one aspect of the present invention. A process 600 is a method for inferring types to enhance flow analysis. The process 600 includes an act 602 for generating constraints from a program and an act 604 for solving the constraints to infer at least one type. The act of solving includes propagating polarities so as to enhance flow analysis. Each polarity is adapted to indicate at least one of a negative polarity, a positive polarity, and a bidirectional polarity. The negative polarity defines that a value is flowing into a type. The positive polarity defines that a value is flowing out of a type. The bidirectional polarity defines a combination of a negative polarity and a positive polarity.

In one embodiment, the act 602 for generating constraints may be mathematically illustrated by the type rules presented following this paragraph. These rules are of the form $A \vdash e: \sigma/C$. σ represents either a location or a type. One interpretation of the form of the rules includes the following: in the type environment A, expression e can be given type or location σ, on the condition of the constraint set C. A type environment A is a set of assignments of the form $x : [\tau]^{\ell}$. Such a form means the inclusion of assigning the location $[\tau]^{\ell}$ to program variable x. The constraint set C includes a set of equalities and inequalities between types, written as $\tau = \tau'$ and $\tau \leq^{index}_{polarity} \tau'$, respectively. An equality $\tau = \tau'$ means the inclusion that the types τ and τ' be selectively unified. An inequality $\tau \leq^{index}_{polarity} \tau'$ means the inclusion that τ' is an instance of τ. Such an inequality is generated whenever type rule [Fun] (discussed below) is applied. τ, in such an inequality, represents a type inferred from the definition of a function f (via rule [Def] which is discussed below). τ' represents the instance type inferred for a particular use of the function f (for example, via rule [Call] which is discussed below). A portion of the type rules are now presented:

$$[\text{Fun}]\frac{\beta \text{ fresh}}{A \vdash f_i : \beta / \{\alpha_f \leq_o^i \beta\}}$$

$$[\text{Var}]\frac{A(x) = [\tau]^\ell}{A \vdash x : [\tau]^\ell / \emptyset}$$

$$[\text{Call}]\frac{\begin{array}{l} A \vdash e_0 : \tau_0 / C_0 \\ A \vdash e_i : \tau_i / C_i (i = 1 \ldots n) \\ C' = \cup_{j=0}^{n} C_j \\ C'' = \{\tau_0 = (\tau_1, \ldots, \tau_n) \to \tau\} \end{array}}{A \vdash e_0(e_1, \ldots, e_n) : \tau / C' \cup C''}$$

$$[\text{Asn}]\frac{\begin{array}{c} A \vdash e_i : [\tau]^\ell / C_1 \\ A \vdash e_2 : \tau', / C_2 \\ C_3 = \{\tau = \tau'\} \end{array}}{A \vdash e_1 = e_2 : \tau' / C_1 \cup C_2 \cup C_3}$$

$$[\text{Rval}]\frac{A \vdash e : [\tau]^\ell / C}{A \vdash e : \tau / C}$$

$$[\text{Addr}]\frac{A \vdash e : [\tau]^\ell / C}{A \vdash \& e : ptr^\ell(\tau) / C}$$

$$[\text{Deref}]\frac{A \vdash e : ptr^\ell(\tau) / C}{A \vdash * e : [\tau]^\ell / C}$$

$$[\text{Cmp}]\frac{\begin{array}{c} A \vdash s_1 : C_1 \\ A \vdash s_2 : C_2 \end{array}}{A \vdash s_1 ; s_2 : C_1 \cup C_2}$$

$$[\text{Local}]\frac{A, x : [\tau]^\ell \vdash s : C}{A \vdash \text{local}\, x \,\text{in}\, s : C}$$

$$[\mathrm{Def}]\frac{\begin{array}{l}A, x_1 : [\tau_1]^{\ell_1}, \ldots, x_n : [\tau_n]^{\ell_n} \vdash s : C \\ C' = C \cup \{\alpha_f = (\tau_1, \ldots \tau_n) \rightarrow^{\ell} \alpha_{ret(f)}\}\end{array}}{A \vdash f(x_1, \ldots, x_n)\{s\} : C'}$$

$$[\mathrm{Ret}]\frac{\begin{array}{l}A \vdash e : \tau / C \\ C' = C \cup \{\alpha_{ret(f)} = \tau\}\end{array}}{A \vdash \mathrm{return}_f\, e : C'}$$

The [Fun] rule is selectively applied whenever the constraint generation encounters a function invocation, such as f_i, in a program. The index i indicates a particular occurrence of the invocation of the function f. The type of the function f is unknown and is given a type variable β. However, β must be a type that is an instantiation of another unknown type α_f. The constraint generation prescribes the polarity of the instantiation as positive (or 0) in such an instantiation. The [Fun] rule incorporates function pointers seamlessly into the analysis.

The act 604 for solving the constraints includes the following constraint closure rule $\tau \leq_p^i \tau_1 \wedge \tau \leq_q^i \tau_2 \Rightarrow \tau_1 = \tau_2$. Notice that the index i is the same on both inequalities. This rule indicates that any two occurrences of the same type variable get instantiated to the same type, within a single instantiation. Such a rule collapses well-matched call/return flow of a flow analysis, such that the computation of flow analysis is enhanced since those well-matched call/return flows need not be considered.

Another aspect of the act 604 includes propagating polarities. Polarities propagate to constraints on subterms of types according to the variance of the subterm. Covariant subterms inherit the polarity from the parent, contravariant subterms obtain the negated polarity of their parent, and non-variant subterms obtain the T polarity. Since subterms of pointer types are non-variant, the polarity

is symbolized as T. Two propagation rules include:

a) $(\tau_1,\ldots\tau_n) \rightarrow^{\ell} \tau \leq_p^i (\tau_1',\ldots\tau_n') \rightarrow^{l'} \tau' \Leftrightarrow \tau_k \leq_{-p}^i \tau_k' \wedge \tau \leq_p^i \tau' \wedge l \leq_p^i l'$

b) $\mathrm{ptr}^{l}(\tau) \leq_p^i \mathrm{ptr}^{l'}(\tau') \Leftrightarrow \tau \leq_{\mathrm{T}}^i \tau' \wedge l \leq_p^i l'$

Negation of polarities is defined as follows: -0 = 1, -1 = 0, and –T = T.

Figure 7 is a structure diagram of a data structure according to one aspect of the present invention. A data structure 700 is used to enhance flow analysis. The data structure 700 includes a data member type 702 to represent a type of a program expression and a data member flow 704 to represent a flow path between two types. The data member type 702 is adapted to contain at least one of a generic type and an instance type. The data member type 702 is adapted to contain an instance type that is an instantiation of a function type.

The data member flow 704 includes a data member polarity. The data member flow 704 also includes a data member index. In one embodiment, the data member flow 704 contains an address of another type so that the data member flow represents a flow path between the data member type and the another type.

Figure 8 is a process diagram of a method according to one aspect of the present invention. A process 800 is a method for forming a graph to enhance flow analysis. The process 800 includes an act for forming a plurality of nodes to represent types. The process 800 includes an act for forming a plurality of instantiation paths to represent instantiations of generic types to instance types. Each instantiation path includes an index and a polarity. The process 800 further includes an act for forming a plurality of flow paths to represent a flow of values between types. Each flow path includes a polarity that is inherited from an instantiation path. The polarity defines at least one of an ingress and an egress flow of value with respect to a type. The index defines an occurrence of an instantiation of a generic type to an instance type so as to differentiate among occurrences of instantiations of the generic type to the instance type.

Conclusion

Methods and structures have been discussed to enhance flow analysis for programs. Such enhancement allows tools such as program optimizers, error detection tools, and user feedback tools to make superior assumptions about programs under analysis. One result from such enhancement includes software products that may run faster, contain fewer bugs, or both. These methods allow a flow analysis to scale well to large programs while providing a desired level of analytical precision within a desired duration of analysis. The methods described hereinbefore comprise a number of acts, but the invention is not so limited since these acts can be performed together as a single act or any combinations of acts.

Although the specific embodiments have been illustrated and described herein, it will be appreciated by those of ordinary skill in the art that any arrangement which is calculated to achieve the same purpose may be substituted for the specific embodiments shown. This application is intended to cover any adaptations or variations of the present invention. It is to be understood that the above description is intended to be illustrative, and not restrictive. Combinations of the above embodiments and other embodiments will be apparent to those of skill in the art upon reviewing the above description. The scope of the invention includes any other applications in which the above structures and fabrication methods are used. Accordingly, the scope of the invention should only be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.

We claim:

1. A method for enhancing flow analysis, comprising:

inferring types from a program;

forming a type graph having polarities and indices from the types; and

forming a flow graph from the type graph to inhibit imprecise paths so as to enhance context-sensitivity of flow analysis.

2. The method of claim 1, wherein inferring includes generating constraints from the program and solving the constraints.

3. The method of claim 2, wherein generating includes generating the constraints, wherein the constraints include a set of equalities and inequalities, wherein the set is a finite set, and wherein the set of equalities and inequalities is adapted to be a set of simultaneous equations.

4. The method of claim 3, wherein generating includes generating the constraints, wherein an equality from the set of equalities defines that a type is equal to another type, and wherein the type and the another type are adapted to be unified.

5. The method of claim 3, wherein generating includes generating the constraints, wherein an inequality from the set of inequalities defines that a type is an instance of another type, wherein the inequality includes an index and a polarity.

6. A computer readable medium having instructions stored thereon for causing a computer to perform a method for enhancing flow analysis, the method comprising:

inferring types from a program;

forming a type graph having polarities and indices from the types; and

forming a flow graph from the type graph so as to enhance context-sensitivity of flow analysis.

7. A method for enhancing flow analysis, comprising:

forming a type graph that includes polarities and indices; and

forming a flow graph that includes a set of flow paths, wherein the set of flow paths excludes imprecise paths so as to enhance context-sensitivity of flow analysis.

8. The method of claim 7, wherein forming the set of flow paths includes forming at least one flow path that inherits a polarity from the polarities of the

type graph.

9. The method of claim 8, wherein forming includes forming the set of flow paths, wherein each imprecise path includes two flow edges, wherein the polarity of one of the two flow edges is negative, and wherein the polarity of the other of the two flow edges is positive.

10. The method of claim 9, wherein forming includes forming the set of flow paths, wherein each imprecise path is defined by the one of the two flow edges preceding the other of the two flow edges.

11. The method of claim 7, wherein forming includes forming the type graph, wherein each polarity indicates at least one of a value flowing into a type and a value flowing out of a type.

12. A computer readable medium having instructions stored thereon for causing a computer to perform a method for enhancing flow analysis, the method comprising:

forming a type graph that includes polarities and indices; and

forming a flow graph that includes a set of flow paths, wherein the set of flow paths excludes imprecise paths so as to enhance context-sensitivity of flow

analysis.

13. A method for forming a graph to enhance flow analysis, comprising:

forming a plurality of nodes to represent types; and

forming a plurality of instantiation paths to represent instantiations of generic types to instance types, wherein each instantiation path includes an index and a polarity.

14. The method of claim 13, further comprising forming a plurality of flow paths to represent a flow of values between types.

15. The method of claim 14, wherein forming includes forming a plurality of flow paths, wherein each flow path includes a polarity that is inherited from an instantiation path.

16. The method of claim 15, wherein forming includes forming a plurality of flow paths, wherein the polarity defines at least one of an ingress and an egress flow of a value with respect to a type.

17. The method of claim 13, wherein forming includes forming a plurality of instantiation paths, wherein the index defines an occurrence of an instantiation of a

generic type to an instance type so as to differentiate among occurrences of instantiations of the generic type to the instance type.

18. A computer readable medium having instructions stored thereon for causing a computer to perform a method for forming a graph to enhance flow analysis, the method comprising:

forming a plurality of nodes to represent types; and

forming a plurality of instantiation paths to represent instantiations of generic types to instance types, wherein each instantiation path includes an index and a polarity.

19. A method for enhancing flow analysis, comprising:

abstracting program expressions into types; and

forming a type graph from the types, wherein the type graph includes polarities and indices so as to enhance flow analysis.

20. The method of claim 19, wherein forming includes forming a plurality of nodes, wherein each node represents a type.

21. The method of claim 20, wherein forming includes forming at least one instantiation path between two nodes to represent an instantiation of a generic type

to an instance type.

22. The method of claim 21, wherein forming includes forming at least one instantiation path that includes an index and a polarity.

23. The method of claim 19, wherein abstracting includes unifying recursive types, wherein unifying is adapted to be executed using a cyclic unification technique.

24. A computer readable medium having instructions stored thereon for causing a computer to perform a method for enhancing flow analysis, the method comprising:

abstracting program expressions into types; and

forming a type graph from the types, wherein the type graph includes polarities and indices so as to enhance flow analysis.

25. A method for inferring types to enhance flow analysis, comprising:

generating constraints from a program; and

solving the constraints to infer at least one type, wherein solving includes propagating polarities so as to enhance flow analysis.

26. The method of claim 25, wherein propagating includes propagating polarities, wherein each polarity is adapted to indicate at least one of a negative polarity, a positive polarity, and a bidirectional polarity.

27. The method of claim 25, wherein propagating includes propagating polarities, wherein the negative polarity defines a value flowing into a type.

28. The method of claim 25, wherein propagating includes propagating polarities, wherein the positive polarity defines a value flowing out of a type.

29. The method of claim 25, wherein propagating includes propagating polarities, wherein the bi-directional polarity defines a combination of a negative polarity and a positive polarity.

30. A computer readable medium having instructions stored thereon for causing a computer to perform a method for inferring types to enhance flow analysis, the method comprising:

generating constraints from a program; and

solving the constraints to infer at least one type, wherein solving includes propagating polarities so as to enhance flow analysis.

31. A method for enhancing flow analysis, comprising:

annotating each expression in a program by a label;

associating the label of an expression with a type of the expression; and

tracing at least one path on a type graph having polarities to determine if a value arising at one label in the program flows to another label in the program.

32. A computer readable medium having instructions stored thereon for causing a computer to perform a method for enhancing flow analysis, the method comprising:

annotating each expression in a program by a label;

associating the label of an expression with a type of the expression; and

tracing at least one path on a type graph having polarities to determine if a value arising at one label in the program flows to another label in the program.

33. A method for enhancing flow analysis, comprising:

forming a type instantiation graph that includes polarities and indices; and

computing points-to information for at least one program point by answering reachability queries on the type instantiation graph.

34. The method of claim 33, wherein forming includes forming with a polymorphic inference technique based on instantiation constraints.

35. The method of claim 33, wherein answering includes answering reachability queries within a desired duration, wherein the desired duration is linearly proportional to a size of the type instantiation graph.

36. A computer readable medium having instructions stored thereon for causing a computer to perform a method for enhancing flow analysis, the method comprising:

forming a type instantiation graph that includes polarities and indices; and

computing points-to information for at least one program point by answering reachability queries on the type instantiation graph.

37. A data structure to enhance flow analysis, comprising:

a data member type to represent a type of a program expression; and

a data member flow having a data member polarity and a data member index to represent a flow path between two types.

38. The data structure of claim 37, wherein the data member type is adapted to contain at least one of a generic type and an instance type.

39. The data structure of claim 37, wherein the data member type is adapted to

contain an instance type that is an instantiation of a function type.

40. The data structure of claim 37, wherein the data member type is adapted to contain an instance type that is an instantiation of a pointer type.

41. The data structure of claim 37, wherein the data member flow contains an address of another type so that the data member flow represents a flow path between the data member type and the another type.

42. A method for enhancing flow analysis, comprising:

forming a type instantiation graph that includes polarities and indices;

forming a flow graph to form a set of flow paths; and

forming a subset from the set of flow paths such that context-sensitivity of flow analysis is enhanced.

43. The method of claim 42, wherein forming includes forming a set of flow paths that includes at least one imprecise path, wherein the subset excludes the at least one imprecise path.

44. The method of claim 43, wherein forming the type-instantiation graph includes forming a plurality of nodes, wherein each node of the plurality of nodes

represents a type expression.

45. The method of claim 44, wherein forming includes forming a plurality of instantiation paths, wherein each instantiation path represents an instantiation of a generic type expression to an instance type expression, wherein each instantiation path includes an index and a polarity, wherein the index represents an occurrence of an instantiation of the generic type expression to the instance type expression.

46. The method of claim 45, wherein forming includes forming a set of flow paths such that each flow path includes a polarity, wherein the polarity of the flow path is positive if the polarity of the instantiation path is positive, and wherein the polarity of the flow path is negative if the polarity of the instantiation path is negative.

47. The method of claim 46, wherein forming a set of flow paths includes forming a path, wherein the path includes at least two flow edges.

48. The method of claim 47, wherein forming a path includes forming a path, wherein the path is defined to be an imprecise path if a polarity of one of the at least two flow edges is negative and a polarity of an other of the at least two flow edges is positive, and wherein the one of the at least two flow edges precedes the

other of the at least two flow edges.

49. A computer readable medium having instructions stored thereon for causing a computer to perform a method for enhancing flow analysis, the method comprising:

forming a type instantiation graph that includes polarities and indices;

forming a flow graph to form a set of flow paths; and

forming a subset from the set of flow paths such that context-sensitivity of flow analysis is enhanced.

50. A graph for enhancing program analysis, the graph comprising:

a plurality of nodes that represent type expressions;

a plurality of instantiation lines that represent type instantiations; and

a plurality of flow lines, wherein each flow line represents a flow direction of at least one value.

51. The graph of claim 50, wherein each instantiation line emanates from one node that is indicative of a generic type expression and ends at another node that is indicative of an instance type expression to define an instantiation direction.

52. The graph of claim 50, wherein each flow line includes a polarity, wherein

each flow line emanates from one node that is indicative of the source of the at least one value and ends at another node that is indicative of the target of the at least one value to define a flow direction.

53. The graph of claim 52, wherein each instantiation line includes an instantiation constraint.

54. The graph of claim 53, wherein the instantiation constraint includes an index and a polarity, wherein the index represents an occurrence of an instantiation of the generic type to the instance type.

55. The graph of claim 54, wherein the polarity of the flow line is positive if the polarity of the instantiation line is positive, wherein the polarity of the flow line is negative if the polarity of the instantiation line is negative.

56. The graph of claim 54, wherein the flow direction of the flow line is the same as the instantiation direction of the instantiation line if the polarity of the instantiation line is positive, wherein the flow direction of the flow line is opposite the instantiation direction of the instantiation line if the polarity of the instantiation line is negative.

METHODS FOR ENHANCING FLOW ANALYSIS

Abstract of the Disclosure

Methods and structures are described that enhance flow analysis for programs. Whereas previous methods are complicated by the presence of function pointers, the present methods present a framework that abstracts function pointers as if they were any other program expressions so as to allow a desired level of analytical decision within a desired duration of analysis. One aspect of the present invention includes inferring types from a program, forming a type graph from the types, and forming a flow graph from the type graph to inhibit imprecise paths so as to enhance context-sensitivity of flow analysis. The methods may be used in any analysis tools such as code browsers and slicing tools.

"Express Mail" mailing label number: EL618476445US
Date of Deposit: JUNE 30, 2000
I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to the Assistant Commissioner for Patents, Washington, D.C. 20231
Printed Name SHAWN L. HISE
Signature

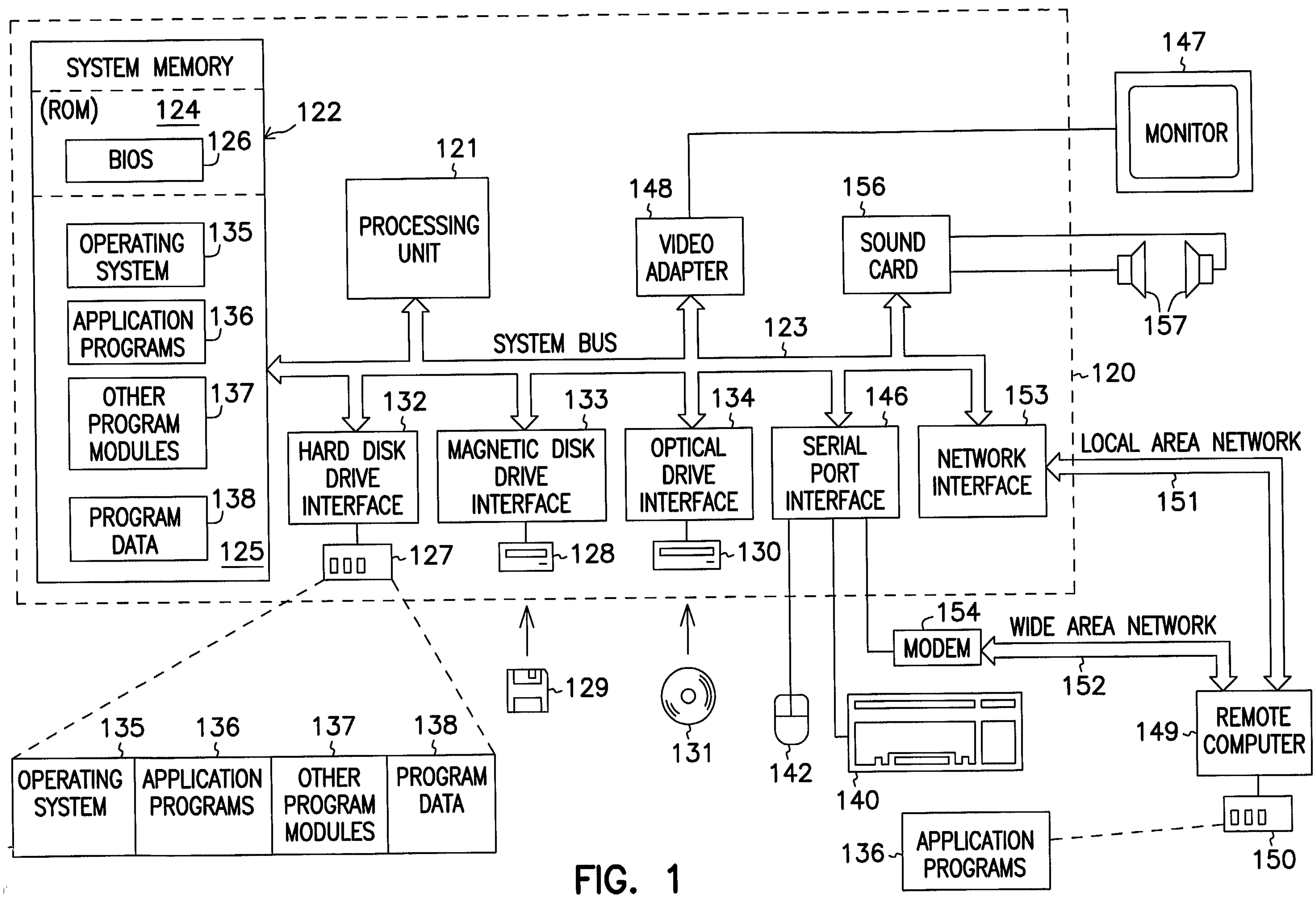


FIG. 1

09609494 .063000

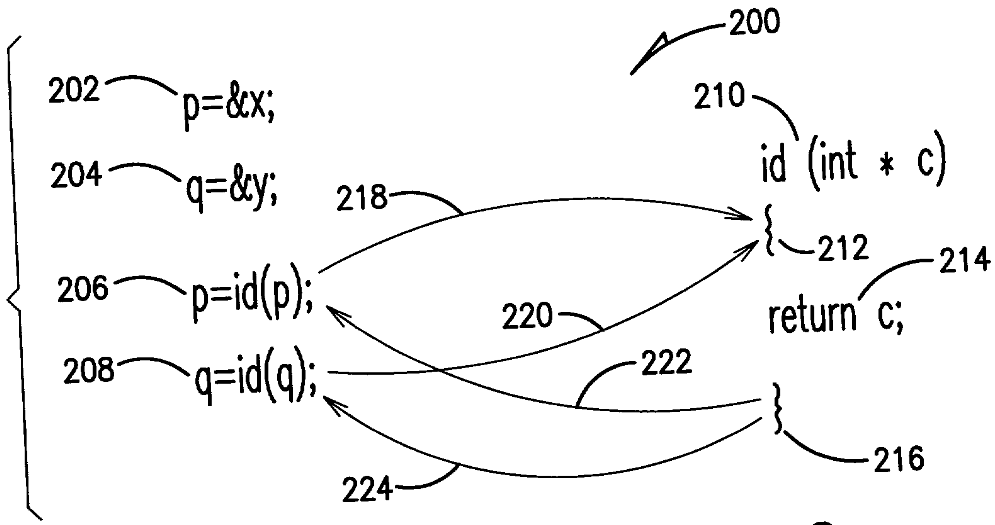

200
202 p=&x;
204 q=&y;
206 p=id(p);
208 q=id(q);
210 id (int * c)
{ 212
return c; 214
} 216
218
220
222
224

FIG. 2

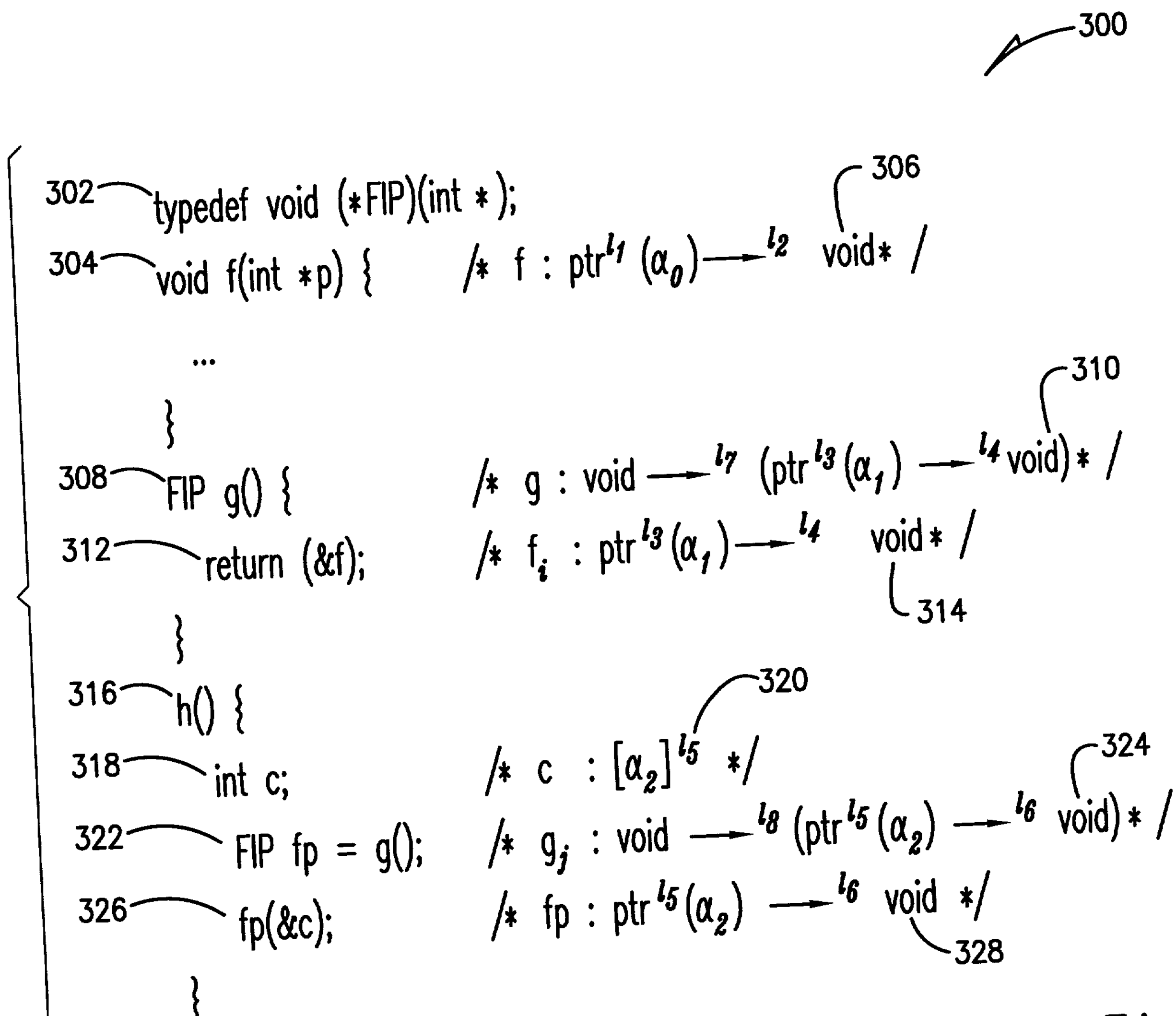

300
302 typedef void (*FIP)(int *);
304 void f(int *p) { /* f : ptr^l1 (α0) →^l2 void* / 306
...
}
308 FIP g() { /* g : void →^l7 (ptr^l3 (α1) →^l4 void) * / 310
312 return (&f); /* fi : ptr^l3 (α1) →^l4 void * / 314
}
316 h() {
318 int c; /* c : [α2]^l5 */ 320
322 FIP fp = g(); /* gj : void →^l8 (ptr^l5 (α2) →^l6 void) * / 324
326 fp(&c); /* fp : ptr^l5 (α2) →^l6 void */ 328
}

FIG. 3A

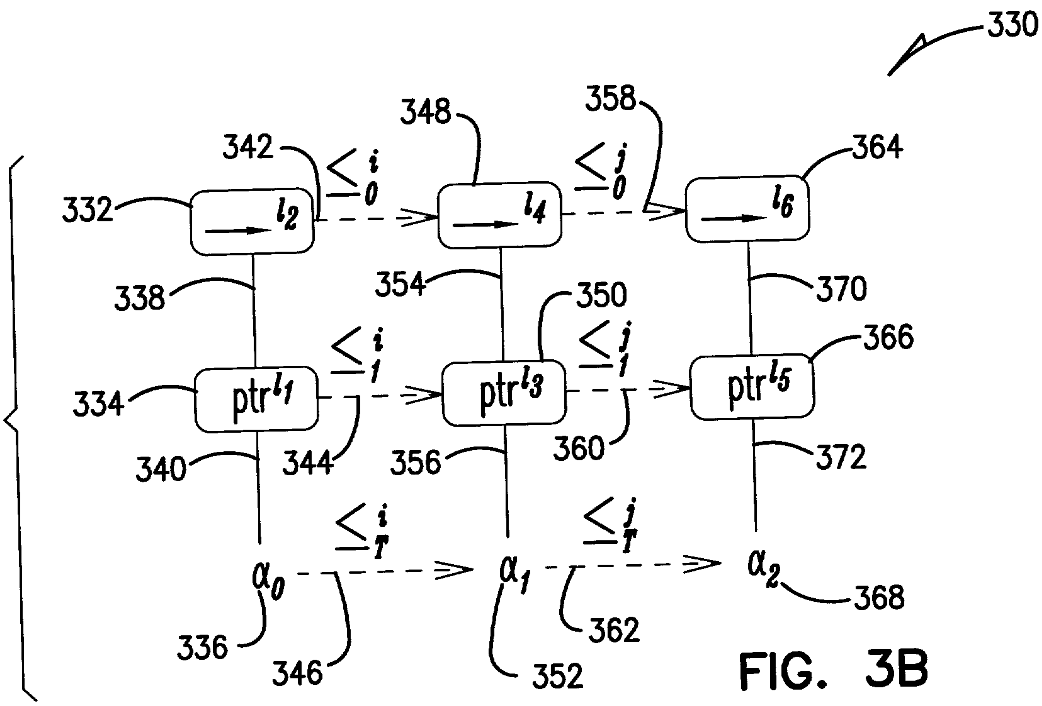

330
332
334
336
338
340
342
344
346
348
350
352
354
356
358
360
362
364
366
368
370
372
l_2
l_4
l_6
ptr^{l_1}
ptr^{l_3}
ptr^{l_5}
α_0
α_1
α_2
$\leq^i_0$
$\leq^j_0$
$\leq^i_1$
$\leq^j_1$
$\leq^i_T$
$\leq^j_T$

FIG. 3B

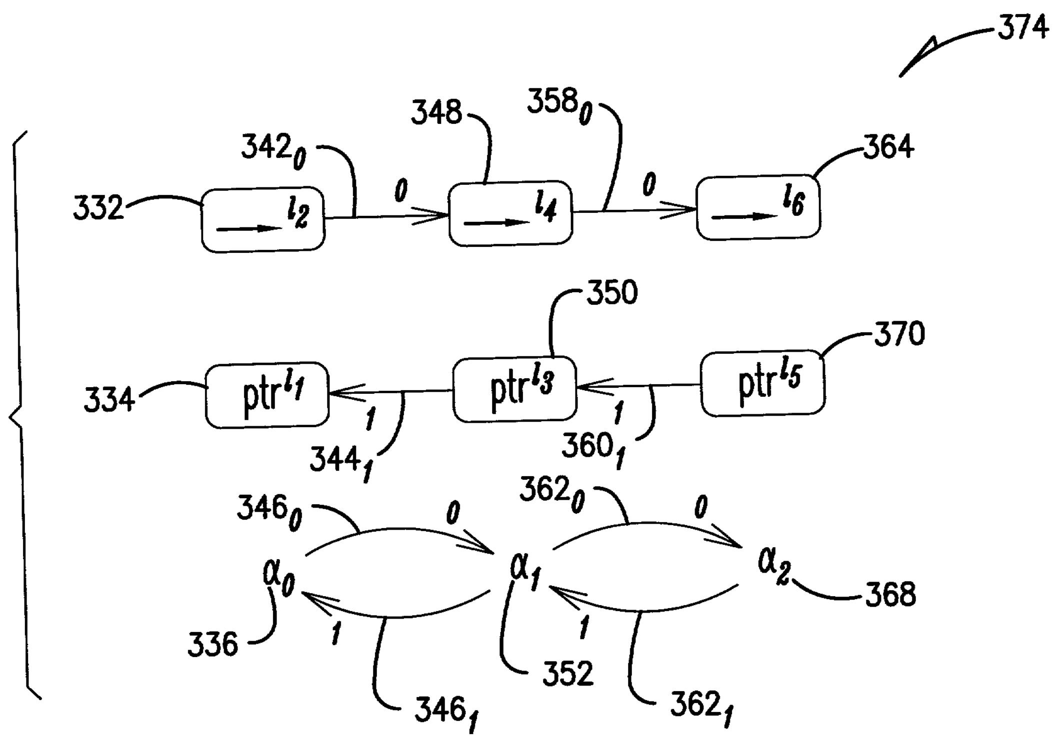
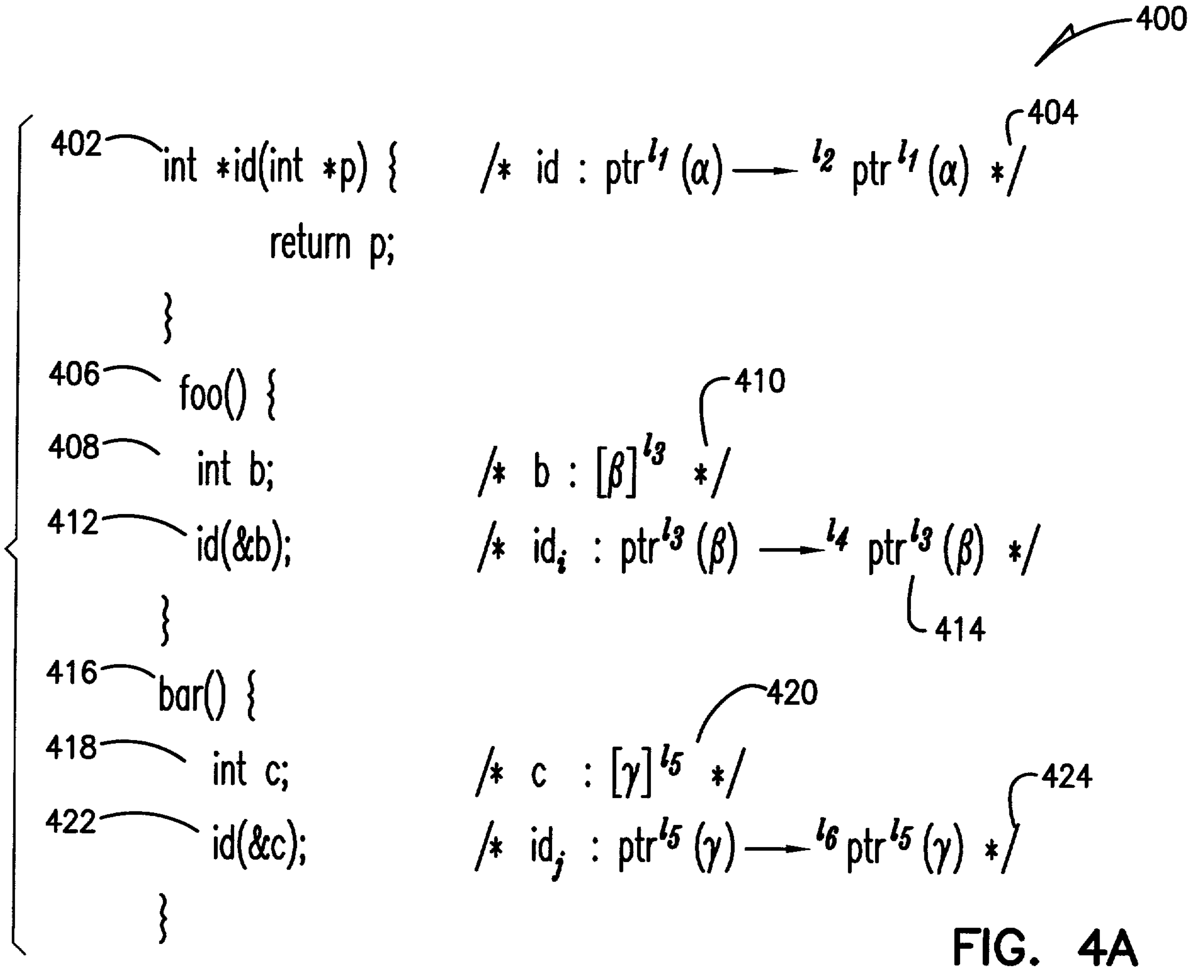
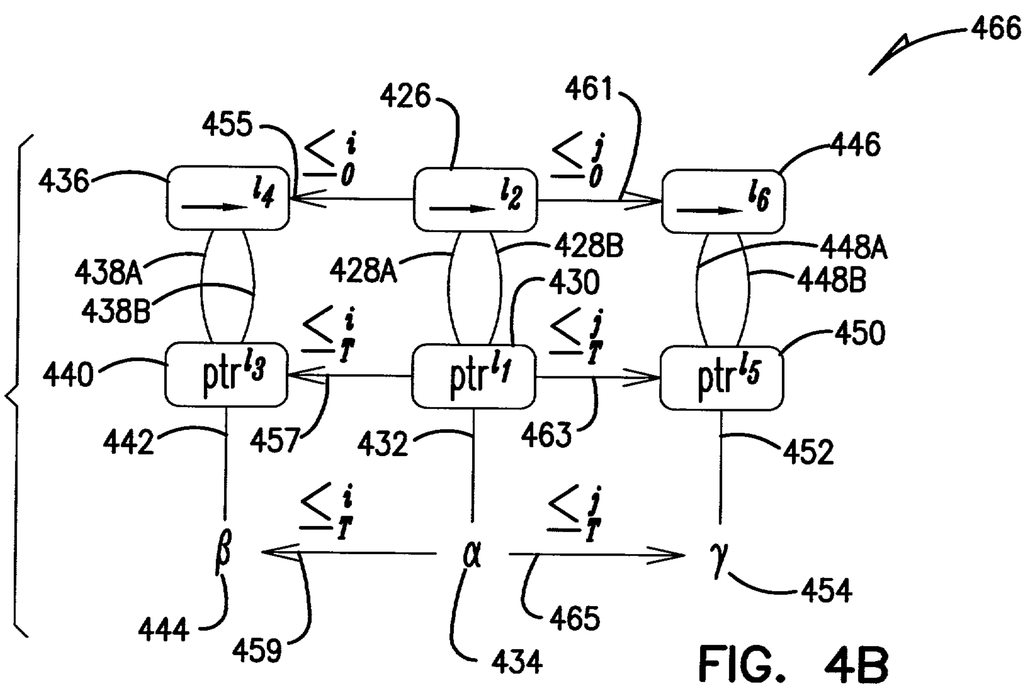

374
332
334
336
342_0
344_1
346_0
346_1
348
350
352
358_0
360_1
362_0
362_1
364
368
370
l_2
l_4
l_6
ptr^{l_1}
ptr^{l_3}
ptr^{l_5}
α_0
α_1
α_2
0
1

FIG. 3C



400
402 int *id(int *p) {
return p;
}
/* id : ptr^l1 (α) —→ ^l2 ptr^l1 (α) */
404
406 foo() {
408 int b;
/* b : [β]^l3 */
410
412 id(&b);
/* id_i : ptr^l3 (β) —→ ^l4 ptr^l3 (β) */
414
}
416 bar() {
418 int c;
/* c : [γ]^l5 */
420
422 id(&c);
/* id_j : ptr^l5 (γ) —→ ^l6 ptr^l5 (γ) */
424
}

FIG. 4A



466
426
461
455
446
436
l4
l2
l6
438A
438B
428A
428B
430
448A
448B
440
ptr^l3
ptr^l1
ptr^l5
450
442
457
432
463
452
β
α
γ
444
459
434
465
454

FIG. 4B

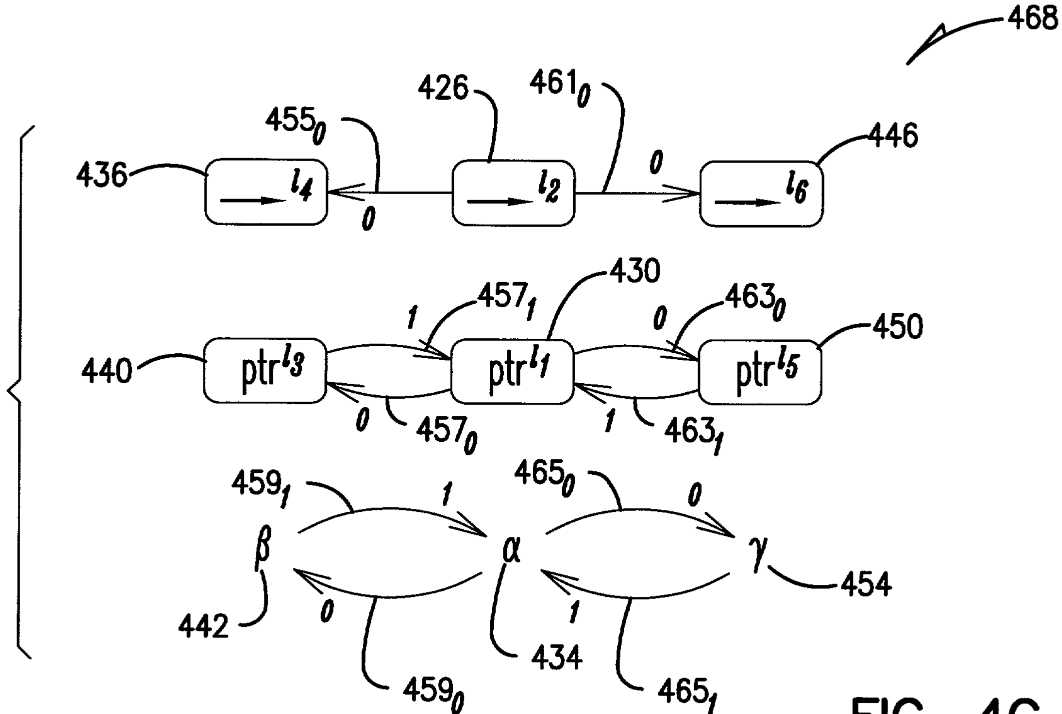


FIG. 4C

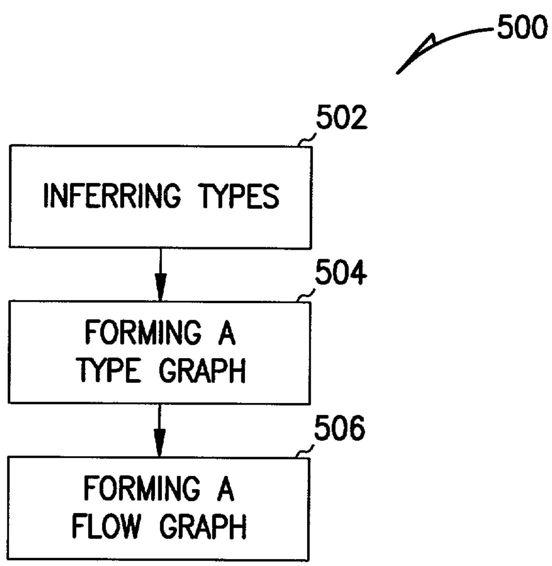


FIG. 5

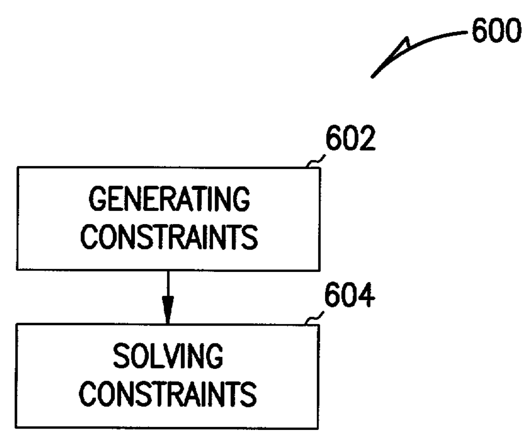


FIG. 6

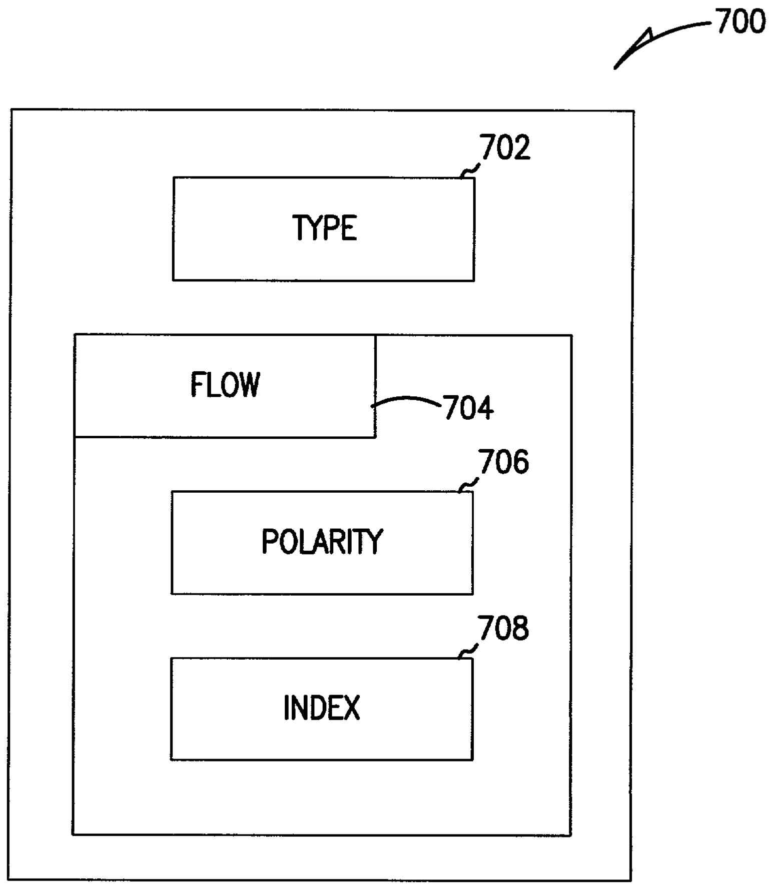

700
702
TYPE
FLOW
704
706
POLARITY
708
INDEX

FIG. 7

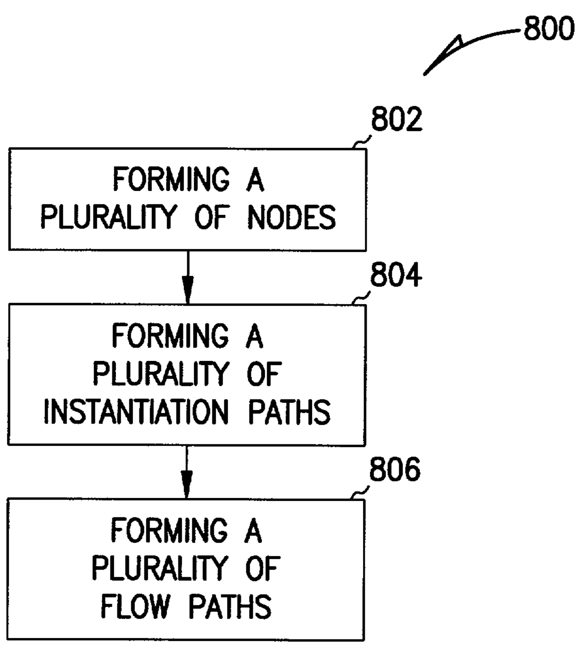

800
802
FORMING A PLURALITY OF NODES
804
FORMING A PLURALITY OF INSTANTIATION PATHS
806
FORMING A PLURALITY OF FLOW PATHS

FIG. 8

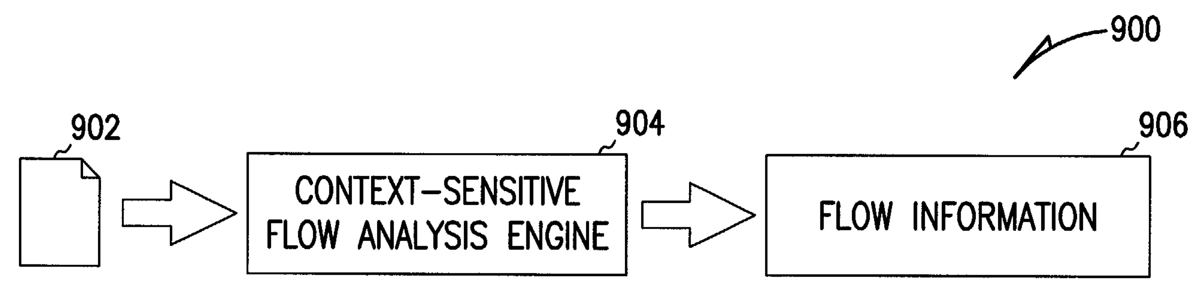

900
902
904
CONTEXT-SENSITIVE
FLOW ANALYSIS ENGINE
906
FLOW INFORMATION

FIG. 9

Attorney Docket No. 777.383US1

SCHWEGMAN LUNDBERG WOESSNER KLUTH

United States Patent Application

COMBINED DECLARATION AND POWER OF ATTORNEY

As a below named inventor I hereby declare that: my residence, post office address and citizenship are as stated below next to my name; that

I verily believe I am the original, first and joint inventor of the subject matter which is claimed and for which a patent is sought on the invention entitled: **METHODS FOR ENHANCING FLOW ANALYSIS**.

The specification of which is attached hereto.

I hereby state that I have reviewed and understand the contents of the above-identified specification, including the claims, as amended by any amendment referred to above.

I acknowledge the duty to disclose information which is material to the patentability of this application in accordance with 37 C.F.R. § 1.56 (attached hereto). I also acknowledge my duty to disclose all information known to be material to patentability which became available between a filing date of a prior application and the national or PCT international filing date in the event this is a Continuation-In-Part application in accordance with 37 C.F.R. §1.63(e).

I hereby claim foreign priority benefits under 35 U.S.C. §119(a)-(d) or 365(b) of any foreign application(s) for patent or inventor's certificate, or 365(a) of any PCT international application which designated at least one country other than the United States of America, listed below and have also identified below any foreign application for patent or inventor's certificate having a filing date before that of the application on the basis of which priority is claimed:

No such claim for priority is being made at this time.

I hereby claim the benefit under 35 U.S.C. § 119(e) of any United States provisional application(s) listed below:

No such claim for priority is being made at this time.

I hereby claim the benefit under 35 U.S.C. § 120 or 365(c) of any United States and PCT international application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States or PCT international application in the manner provided by the first paragraph of 35 U.S.C. § 112, I acknowledge the duty to disclose material information as defined in 37 C.F.R. § 1.56(a) which became available between the filing date of the prior application and the national or PCT international filing date of this application:

No such claim for priority is being made at this time.

I hereby appoint the following attorney(s) and/or patent agent(s) to prosecute this application and to transact all business in the Patent and Trademark Office connected herewith:

Anglin, J. Michael	Reg. No. 24,916	Huebsch, Joseph C.	Reg. No. 42,673	Nelson, Albin J.	Reg. No. 28,650
Bianchi, Timothy E.	Reg. No. 39,610	Jurkovich, Patti J.	Reg. No. 44,813	Nielsen, Walter W.	Reg. No. 25,539
Billion, Richard E.	Reg. No. 32,836	Kalis, Janal M.	Reg. No. 37,650	Oh, Allen J.	Reg. No. 42,047
Black, David W.	Reg. No. 42,331	Kaufmann, John D.	Reg. No. 24,017	Padys, Danny J.	Reg. No. 35,635
Brennan, Leoniede M.	Reg. No. 35,832	Klima-Silberg, Catherine I.	Reg. No. 40,052	Parker, J. Kevin	Reg. No. 33,024
Brennan, Thomas F.	Reg. No. 35,075	Kluth, Daniel J.	Reg. No. 32,146	Perdok, Monique M.	Reg. No. 42,989
Brooks, Edward J., III	Reg. No. 40,925	Lacy, Rodney L.	Reg. No. 41,136	Prout, William F.	Reg. No. 33,995
Chu, Dinh C.P.	Reg. No. 41,676	Lemaire, Charles A.	Reg. No. 36,198	Sako, Katie E.	Reg. No. 32,628
Clark, Barbara J.	Reg. No. 38,107	LeMoine, Dana B.	Reg. No. 40,062	Schumm, Sherry W.	Reg. No. 39,422
Crouse, Daniel D.	Reg. No. 32,022	Lundberg, Steven W.	Reg. No. 30,568	Schwegman, Micheal L.	Reg. No. 25,816
Dahl, John M.	Reg. No. 44,639	Mack, Lisa K.	Reg. No. 42,825	Smith, Michael G.	Reg. No. 45,368
Drake, Eduardo E.	Reg. No. 40,594	Maeyaert, Paul L.	Reg. No. 40,076	Speier, Gary J.	Reg. No. 45,458
Embretson, Janet E.	Reg. No. 39,665	Maki, Peter C.	Reg. No 42,832	Steffey, Charles E.	Reg. No. 25,179
Fordenbacher, Paul J.	Reg. No. 42,546	Malen, Peter L.	Reg. No. 44,894	Terry, Kathleen R.	Reg. No. 31,884
Forrest, Bradley A.	Reg. No. 30,837	Mates, Robert E.	Reg. No 35,271	Tong, Viet V.	Reg. No. 45,416
Gamon, Owen J.	Reg. No. 36,143	McCrackin, Ann M.	Reg. No. 42,858	Viksnins, Ann S.	Reg. No. 37,748
Harris, Robert J.	Reg. No. 37,346	Nama, Kash	Reg. No. 44,255	Woessner, Warren D.	Reg. No. 30,440

I hereby authorize them to act and rely on instructions from and communicate directly with the person/assignee/attorney/firm/organization/who/which first sends/sent this case to them and by whom/which I hereby declare that I have consented after full disclosure to be represented unless/until I instruct Schwegman, Lundberg, Woessner & Kluth, P.A. to the contrary.

Please direct all correspondence in this case to **Schwegman, Lundberg, Woessner & Kluth, P.A.** at the address indicated below:

P.O. Box 2938, Minneapolis, MN 55402
Telephone No. (612)373-6900

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Full Name of joint inventor number 1 : **Manuvir Das**
Citizenship: **India** Residence: **Kirkland, WA**
Post Office Address: 13384 NE 134th Place
Kirkland, WA 98034

Signature: ______ Date: 29 June 2000
Manuvir Das

Full Name of joint inventor number 2 : **Manuel A. Fahndrich**
Citizenship: **Switzerland** Residence: **Bellevue, WA**
Post Office Address: 2048 154th Avenue SE
Bellevue, WA 98007

Signature: ______ Date: 29 June 2000
Manuel A. Fahndrich

X Additional inventors are being named on separately numbered sheets, attached hereto.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Full Name of joint inventor number 3 : **Jakob Rehof**
Citizenship: **Denmark** Residence: **Redmond, WA**
Post Office Address: 9120 168th Place NE
Redmond, WA 98052

Signature: Jakob Rehof Date: 6/29/2000
Jakob Rehof

Full Name of inventor:
Citizenship: Residence:
Post Office Address:

Signature: Date:

Full Name of inventor:
Citizenship: Residence:
Post Office Address:

Signature: Date:

Full Name of inventor:
Citizenship: Residence:
Post Office Address:

Signature: Date:

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

§ 1.56 Duty to disclose information material to patentability.

(a) A patent by its very nature is affected with a public interest. The public interest is best served, and the most effective patent examination occurs when, at the time an application is being examined, the Office is aware of and evaluates the teachings of all information material to patentability. Each individual associated with the filing and prosecution of a patent application has a duty of candor and good faith in dealing with the Office, which includes a duty to disclose to the Office all information known to that individual to be material to patentability as defined in this section. The duty to disclose information exists with respect to each pending claim until the claim is canceled or withdrawn from consideration, or the application becomes abandoned. Information material to the patentability of a claim that is canceled or withdrawn from consideration need not be submitted if the information is not material to the patentability of any claim remaining under consideration in the application. There is no duty to submit information which is not material to the patentability of any existing claim. The duty to disclose all information known to be material to patentability is deemed to be satisfied if all information known to be material to patentability of any claim issued in a patent was cited by the Office or submitted to the Office in the manner prescribed by §§ 1.97(b)-(d) and 1.98. However, no patent will be granted on an application in connection with which fraud on the Office was practiced or attempted or the duty of disclosure was violated through bad faith or intentional misconduct. The Office encourages applicants to carefully examine:

(1) prior art cited in search reports of a foreign patent office in a counterpart application, and

(2) the closest information over which individuals associated with the filing or prosecution of a patent application believe any pending claim patentably defines, to make sure that any material information contained therein is disclosed to the Office.

(b) Under this section, information is material to patentability when it is not cumulative to information already of record or being made of record in the application, and

(1) It establishes, by itself or in combination with other information, a prima facie case of unpatentability of a claim; or

(2) It refutes, or is inconsistent with, a position the applicant takes in:

(i) Opposing an argument of unpatentability relied on by the Office, or

(ii) Asserting an argument of patentability.

A prima facie case of unpatentability is established when the information compels a conclusion that a claim is unpatentable under the preponderance of evidence, burden-of-proof standard, giving each term in the claim its broadest reasonable construction consistent with the specification, and before any consideration is given to evidence which may be submitted in an attempt to establish a contrary conclusion of patentability.

(c) Individuals associated with the filing or prosecution of a patent application within the meaning of this section are:

(1) Each inventor named in the application:

(2) Each attorney or agent who prepares or prosecutes the application; and

(3) Every other person who is substantively involved in the preparation or prosecution of the application and who is associated with the inventor, with the assignee or with anyone to whom there is an obligation to assign the application.

(d) Individuals other than the attorney, agent or inventor may comply with this section by disclosing information to the attorney, agent, or inventor.